Map Reduce

Experiment summary

Wu Qingyan

213221715@seu.edu.cn

May 20, 2024

Contents

What is MapReduce

Introduction

- MapReduce is a programming model for large-scale data processing.
- Originally introduced by Google engineers.
- Enables parallel computation on distributed clusters.
- Developers focus on writing small code snippets.

Contents

Building a Simplified MapReduce

Introduction

- In this project, we will construct a simplified version of MapReduce for a single machine.The challenge is how to implement correct concurrency support.This task has three specific objectives:
  1. Understand the general properties of the MapReduce paradigm.
  2. Implement a correct and efficient MapReduce framework using threads and relevant functions.
  3. Gain more experience in writing concurrent code.

Building a Simplified MapReduce

A Simple Example: Word Count

- Let's consider a straightforward yet meaningful example of a word count program based on this framework:

```c
void Map(char *file_name) {
    FILE *fp = fopen(file_name, "r");
    assert(fp != NULL);
    char *line = NULL;
    size_t size = 0;
    while (getline(&line, &size, fp) != -1) {
        char *token, *dummy = line;
        while ((token = strsep(&dummy, " \t\n\r"))
                != NULL) {
            MR_Emit(token, "1");
        }
    }
    free(line);
    fclose(fp);
}
```

```c
void Reduce(char *key, Getter get_next,
        int partition_number) {
    int count = 0;
    char *value;
    while ((value = get_next(key,
            partition_number)) != NULL)
        count++;
    printf("%s %d\n", key, count);
}

int main(int argc, char *argv[]) {
    MR_Run(argc, argv, Map, 10, Reduce,
            10, MR_DefaultHashPartition
           );
}
```

Building a Simplified MapReduce

Components of the MapReduce Framework

- The MapReduce framework consists of the following components:
  - **Map**: Processes input data and emits intermediate key-value pairs.
  - **Reduce**: Aggregates intermediate values for each key.
  - **MR_Run**: Initializes the MapReduce framework and runs the Map and Reduce functions.
  - **MR_Emit**: Emits intermediate key-value pairs.
  - **MR_GetNext**: Retrieves the next value for a given key.
  - **MR_DefaultHashPartition**: Partitions the intermediate key-value pairs.

東南大學 SOUTHEAST UNIVERSITY

Building a Simplified MapReduce

Function Definitions

- In the provided **mapreduce.h** file, we define the following function pointers:

```c
#ifndef __mapreduce_h__
#define __mapreduce_h__

// Different function pointer types used by MR
typedef char *(*Getter)(char *key, int partition_number);
typedef void (*Mapper)(char *file_name);
typedef void (*Reducer)(char *key, Getter get_func, int partition_number);
typedef unsigned long (*Partitioner)(char *key, int num_partitions);

// External functions: these are what you must define

void MR_Emit(char *key, char *value);

unsigned long MR_DefaultHashPartition(char *key, int num_partitions);

void MR_Run(int argc, char *argv[],
            Mapper map, int num_mappers,
            Reducer reduce, int num_reducers,
            Partitioner partition);

#endif // __mapreduce_h__
```

- In the following sections of this presentation, I will sequentially implement the function pointers mentioned in the **mapreduce.h** file.

东南大學 SOUTHEAST UNIVERSITY

Building a Simplified MapReduce

Data Structure

- Before we look at the implementation of the functions, let's define the data structure that will be used in the MapReduce framework.

```c
typedef struct KeyValuePair {
    char *key;
    char *value;
    int partition;
    struct KeyValuePair *next;
} KeyValuePair;
```

- The **KeyValuePair** structure contains the following fields:
    - **key**: The key of the key-value pair.
    - **value**: The value of the key-value pair.
    - **partition**: The partition number of the key-value pair.
    - **next**: A pointer to the next key-value pair in the list.
- This structure will be used to store the intermediate key-value pairs emitted by the **MR_Emit** function.

Building a Simplified MapReduce

MR_Emit

```c
void MR_Emit(char *key, char *value) {
    if (key == NULL || strlen(key) == 0) {
        return;
    }

    int partition = global_partition(key, num_partitions);
    KeyValuePair *newPair = (KeyValuePair *)malloc(sizeof(KeyValuePair));
    newPair->key = strdup(key);
    newPair->value = strdup(value);
    newPair->partition = partition;

    pthread_mutex_lock(&lock_emit[partition]);
    insert_sorted(&heads[partition], newPair);
    pthread_mutex_unlock(&lock_emit[partition]);
}
```

- Here we use a global function **global_partition** to determine the partition number for the key-value pair.As we will see later, this function is initialized in the **MR_Run** function.
- The **insert_sorted** function is used to insert the key-value pair into the linked list in sorted order.
- The **lock_emit** array is used to lock the partition when inserting the key-value pair, as many **Map** threads may be writing to the same partition.

Building a Simplified MapReduce

insert_sorted

```c
void insert_sorted(KeyValuePair **head, KeyValuePair *newPair) {
    if (*head == NULL || strcmp(newPair->key, (*head)->key) < 0) {
        newPair->next = *head;
        *head = newPair;
    } else {
        KeyValuePair *current = *head;
        while (current->next != NULL && strcmp(newPair->key, current->next->key) > 0) {
            current = current->next;
        }
        newPair->next = current->next;
        current->next = newPair;
    }
}
```

- The **insert_sorted** function inserts the key-value pair into the linked list in sorted order based on the key.
- This process is similar to insertion sort.
- The time complexity of this function is $O(n)$, where n is the number of elements in the list.

SOUTHEAST UNIVERSITY 東南大學

Building a Simplified MapReduce

MR_DefaultHashPartition

```c
unsigned long MR_DefaultHashPartition(char *key, int num_partitions) {
    unsigned long hash = 5381;
    int c;
    while ((c = *key++) != '\0')
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash % num_partitions;
}
```

- The **MR_DefaultHashPartition** function is a simple hash function that returns the partition number for the key-value pair.

Building a Simplified MapReduce

map_thread

```c
typedef struct {
    char **filenames;
    int num_files;
    Mapper mapper;
} MapArg;

int current_file_index = 0;
```

- The struct above is used to pass arguments to the **map_thread** function
- The **current_file_index** variable is used to keep track of the current processing file.

```c
void *map_thread(void *arg) {
    MapArg *mapArg = (MapArg *)arg;

    while (1) {
        pthread_mutex_lock(&lock_index);
        if (current_file_index >= mapArg->num_files) {
            pthread_mutex_unlock(&lock_index);
            break;
        }
        char *filename = mapArg->filenames[
                current_file_index++];
        pthread_mutex_unlock(&lock_index);

        mapArg->mapper(filename);
    }

    return NULL;
}
```

Building a Simplified MapReduce

map_thread.Cond

```c
void *map_thread(void *arg) {
    MapArg *mapArg = (MapArg *)arg;

    while (1) {
        pthread_mutex_lock(&lock_index);
        if (current_file_index >= mapArg->num_files) {
            pthread_mutex_unlock(&lock_index);
            break;
        }
        char *filename = mapArg->filenames[current_file_index++];
        pthread_mutex_unlock(&lock_index);

        mapArg->mapper(filename);
    }

    return NULL;
}
```

- The **map_thread** function reads the filenames from the **MapArg** struct and processes each file using the **mapper** function.
- A lock is used to ensure that only one thread can access the **current_file_index** variable at a time.
- When all files have been processed, the thread exits the loop and returns.

Building a Simplified MapReduce

reduce_thread

```
typedef struct {
    int partition_number;
    Reducer reducer;
} ReduceArg;
```

- similar to **MapArg**, the **ReduceArg** struct is used to pass arguments to **reduce_thread**.
- The **reduce_thread** function is relatively long because of freeing memory.

```
void *reduce_thread(void *arg) {
    ReduceArg *reduceArg = (ReduceArg *)arg;
    currents[reduceArg->partition_number] = heads[reduceArg->partition_number];
    while (currents[reduceArg->partition_number] != NULL) {
        char *key = currents[reduceArg->partition_number]->key;
        reduceArg->reducer(key, get_next, reduceArg->partition_number);
    }

    KeyValuePair *current = heads[reduceArg->partition_number];
    while (current != NULL) {
        KeyValuePair *next = current->next;
        free(current->key);
        free(current->value);
        free(current);
        current = next;
    }

    free(arg);
    return NULL;
}
```

東南大學 SOUTHEAST UNIVERSITY

Building a Simplified MapReduce

reduce_thread.Cond

```c
void *reduce_thread(void *arg) {
    ReduceArg *reduceArg = (ReduceArg *)arg;
    currents[reduceArg->partition_number] =
            heads[reduceArg->partition_number
            ];
    while (currents[reduceArg->
            partition_number] != NULL) {
        char *key = currents[reduceArg->
                partition_number]->key;
        reduceArg->reducer(key, get_next,
                reduceArg->partition_number);
    }

    KeyValuePair *current = heads[reduceArg
            ->partition_number];
    while (current != NULL) {
        KeyValuePair *next = current->next;
        free(current->key);
        free(current->value);
        free(current);
        current = next;
    }

    free(arg);
    return NULL;
}
```

- The **reduce_thread** function processes the key-value pairs in the partition and calls the **reducer** function for each key.
- Each thread handles a different partition, so there is no need for locks.
- After processing all key-value pairs, the memory allocated for the key-value pairs is freed.

Building a Simplified MapReduce

get_next

- In the reducer function written by our user, **get_next** function will be called to get the next value for a given key. When there is no more values for the key, or when the partition is empty, the function will return **NULL**.

```
char *get_next(char *key, int partition_number) {
    if (currents[partition_number] != NULL && strcmp(currents[partition_number]->key, key) ==
        0) {
        char *value = currents[partition_number]->value;
        currents[partition_number] = currents[partition_number]->next;
        return value;
    }
    return NULL;
}
```

- Since we have already sorted the key-value pairs in the partition, we can simply traverse the linked list to get the next value for a given key.

Building a Simplified MapReduce

global variables

```
KeyValuePair **heads;          // Array of head pointers for the buckets
KeyValuePair **currents;       // Array of pointers to the current processing key-value pair
int num_partitions;            // Number of partitions
Partitioner global_partition;  // Partition function
pthread_mutex_t *lock_emit;    // Mutex lock for protecting emit operations
pthread_mutex_t lock_index;    // Mutex lock for protecting file index
```

- The global variables above are used in the MapReduce framework.
- Most of them are initialized in the **MR_Run** function, and they are used in the **MR_Emit**, **map_thread** and **reduce_thread** functions.

Building a Simplified MapReduce

MR_Run

```c
void MR_Run(int argc, char *argv[],
            Mapper map, int num_mappers,
            Reducer reduce, int num_reducers,
            Partitioner partition) {
    pthread_t *threads_map = (pthread_t *)malloc(sizeof(pthread_t) * num_mappers);
    pthread_t *threads_reduce = (pthread_t *)malloc(sizeof(pthread_t) * num_reducers);
    lock_emit = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t) * num_reducers);
    for (int i = 0; i < num_reducers; i++) {
        pthread_mutex_init(&lock_emit[i], NULL);
    }
    pthread_mutex_init(&lock_index, NULL);

    num_partitions = num_reducers;
    heads = (KeyValuePair **)malloc(sizeof(KeyValuePair *) * num_partitions);
    for (int i = 0; i < num_partitions; i++) {
        heads[i] = NULL;
    }

    global_partition = partition;
    // to be continued...
```

東南大學 SOUTHEAST UNIVERSITY

Building a Simplified MapReduce

MR_Run.Cond

```
MapArg arg;
arg.filenames = argv + 1;
arg.num_files = argc - 1;
arg.mapper = map;

for (int i = 0; i < num_mappers; i++) {
    pthread_create(&threads_map[i], NULL, map_thread, &arg);
}

for (int i = 0; i < num_mappers; i++) {
    pthread_join(threads_map[i], NULL);
}
current_file_index = 0;

currents = (KeyValuePair **)malloc(sizeof(KeyValuePair *) * num_partitions);
for (int i = 0; i < num_partitions; i++) {
    currents[i] = heads[i];
}

for (int i = 0; i < num_reducers; i++) {
    // printf("MR_Run: starting reducer %d\n", i);
    ReduceArg *arg = (ReduceArg *)malloc(sizeof(ReduceArg));
    arg->partition_number = i;
    arg->reducer = reduce;
    pthread_create(&threads_reduce[i], NULL, reduce_thread, arg);
}

for (int i = 0; i < num_reducers; i++) {
    pthread_join(threads_reduce[i], NULL);
}
\\ to be continued...
```

東南大學 SOUTHEAST UNIVERSITY

Building a Simplified MapReduce
MR_Run.Cond

```c
        for (int i = 0; i < num_reducers; i++) {
            pthread_mutex_destroy(&lock_emit[i]);
        }
        free(lock_emit);
        pthread_mutex_destroy(&lock_index);
        free(threads_map);
        free(threads_reduce);
        free(heads);
        free(currents);
    }
}
```

- This is the end of **MapReduce.c**, in next part we will test the correctness and efficiency of the MapReduce framework.

Contents

Testing and Analysis
Introduction

- We will evaluate our code from three perspectives:
  - **Memory Management**
  - **Single Count Correctness** We need to verify if the word count is correct.
  - **Efficiency of Multithreaded Map and Reduce** We need to check if multithreaded Map and Reduce can truly improve efficiency.

Testing and Analysis

Memory Management

- We will use Valgrind to check for memory leaks in our code.
- We will run the word count program on a small dataset and check for memory leaks.
- The dataset we are using is a collection of short English jokes from `https://github.com/taivop/joke-dataset`.

```
$ sudo apt-get install valgrind
$ gcc -o wordcount wordcount.c mapreduce.c -lpthread
$ valgrind --leak-check=full ./wordcount ./tests/1.in
```

- The proper output should be "All heap blocks were freed -- no leaks are possible".

```
==217449==
==217449== HEAP SUMMARY:
==217449==     in use at exit: 0 bytes in 0 blocks
==217449==   total heap usage: 985 allocs, 985 frees, 23,725 bytes allocated
==217449==
==217449== All heap blocks were freed -- no leaks are possible
==217449==
==217449== For lists of detected and suppressed errors, rerun with: -s
==217449== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

东南大学 SOUTHEAST UNIVERSITY

Testing and Analysis

Word Count Correctness

- We will use the following script to compare the output of our word count program with the expected output.

```
word_count() {
    awk '{for(i=1;i<=NF;i++) wc[$i]++} END {for(word in wc) print word, wc[word]}' $@ | sort
}
# Run the wordcount program and calculate execution time
start_time=$(date +%s.%N)
./wordcount tests/1.in > tests-out/1.out
end_time=$(date +%s.%N)
wordcount_time=$(echo "$end_time - $start_time" | bc)

# Use the word_count function to process the file and save the result
word_count tests/1.in > tests/1.out

# Sort the output of the wordcount program
sort tests-out/1.out -o tests-out/1.out

# Compare the results of the wordcount program and the word_count function
if diff -q tests-out/1.out tests/1.out > /dev/null; then
    result=$(tput setaf 2)correct$(tput sgr0) # Green "correct"
else
    result=$(tput setaf 1)incorrect$(tput sgr0) # Red "incorrect"
fi

# Print the result
printf "test1 %-10s %.4f s\n" $result $wordcount_time
```

Testing and Analysis

Efficiency of Multithreaded Map and Reduce

- First, we will adjust the code in **wordcount.c** to allow it to adjust the number of map threads and reduce threads based on external parameters.

```c
int main(int argc, char *argv[]) {
    int map_num = 10;
    int reduce_num = 10;
    char **new_argv = malloc(sizeof(char *) * argc);
    int new_argc = 0;

    for (int i = 0; i < argc; i++) {
        if (strcmp(argv[i], "--map") == 0 && i + 1 < argc) {
            map_num = atoi(argv[++i]);
        } else if (strcmp(argv[i], "--reduce") == 0 && i + 1 < argc) {
            reduce_num = atoi(argv[++i]);
        } else {
            new_argv[new_argc++] = argv[i];
        }
    }

    MR_Run(new_argc, new_argv, Map, map_num, Reduce, reduce_num, MR_DefaultHashPartition);
    free(new_argv);
}
```

東南大學 SOUTHEAST UNIVERSITY

Testing and Analysis

Efficiency of Multithreaded Map and Reduce.Cond

- Then, we adjust the test script, run the script on a longer text (containing 4000 lines, more than 250,000 words), check the correctness and record the time.

```
for i in {1..4}
do
    map_values=(1 10 100)
    reduce_values=(1 10 100)

    if [ $i -lt 3 ]; then
        # Run the wordcount program and calculate execution time
        start_time=$(date +%s.%N)
        ./wordcount tests/$i.in > tests-out/$i.out
        end_time=$(date +%s.%N)
        wordcount_time=$(echo "$end_time - $start_time" | bc)

        # Use the word_count function to process the file and save the result
        word_count tests/$i.in > tests/$i.out

        # Sort the output of the wordcount program
        sort tests-out/$i.out -o tests-out/$i.out

        # Compare the results of the wordcount program and the word_count function
        if diff -q tests-out/$i.out tests/$i.out > /dev/null; then
            result=$(tput setaf 2)correct$(tput sgr0)  # Green "correct"
        else
            result=$(tput setaf 1)incorrect$(tput sgr0)  # Red "incorrect"
        fi
        # Print the result
        printf "test%-2s %-10s %.4f s\n" $i $result $wordcount_time
    fi
done
```

Testing and Analysis

Efficiency of Multithreaded Map and Reduce.Cond

```
for i in {1..4}
do
    map_values=(1 10 100)
    reduce_values=(1 10 100)
    if [ $i -lt 3 ]; then
        # Same as before
    else
        # Get all files
        files=$(find tests/$i.in -type f -name '*.in')

        # Print the header
        printf "%-10s" "test$i"
        for reduce_value in ${reduce_values[@]}
        do
            printf "%-10s" "reduce=$reduce_value"
        done
        printf "\n"
```

```
        # Test each map parameter value once
        for map_value in ${map_values[@]}
        do
            printf "%-10s" "map=$map_value"
            for reduce_value in ${reduce_values[@]}
            do
                # Run the wordcount program and calculate execution time
                start_time=$(date +%s.%N)
                ./wordcount --map $map_value --reduce $reduce_value $files | sort > tests-out/$i-$map_value-$reduce_value.out
                end_time=$(date +%s.%N)
                wordcount_time=$(echo "$end_time - $start_time" | bc)

                # Print the result
                printf "%-10.4f" $wordcount_time
            done
            printf "\n"
        done
    fi
done
```

- Finally, we divide the long text test data into 300 and 100 parts respectively, as test 3 and test 4, and modify the test script to run tests 3 and 4 nine times each

Testing and Analysis

Efficiency of Multithreaded Map and Reduce.Cond

- The results of the test script are shown below:

```
test1   correct 0.0044 s
test2   correct 1.0833 s
test3     reduce=1  reduce=10 reduce=100
map=1     12.9763   1.1909    0.1264
map=10    11.6877   0.5096    0.0671
map=100   19.8852   0.8303    0.1253
test4     reduce=1  reduce=10 reduce=100
map=1     11.1390   1.0024    0.1432
map=10    11.5707   0.4578    0.0746
map=100   20.8295   0.9527    0.0847
```

- First, it's important to note that when the number of reduce threads is 1, increasing the number of map threads can actually decrease efficiency.
- Second, increasing the number of reduce threads can always improve performance significantly.
- Finally, although the hash function can evenly distribute key-value pairs to each partition, considering the different word frequencies, the load on each node may not be balanced.